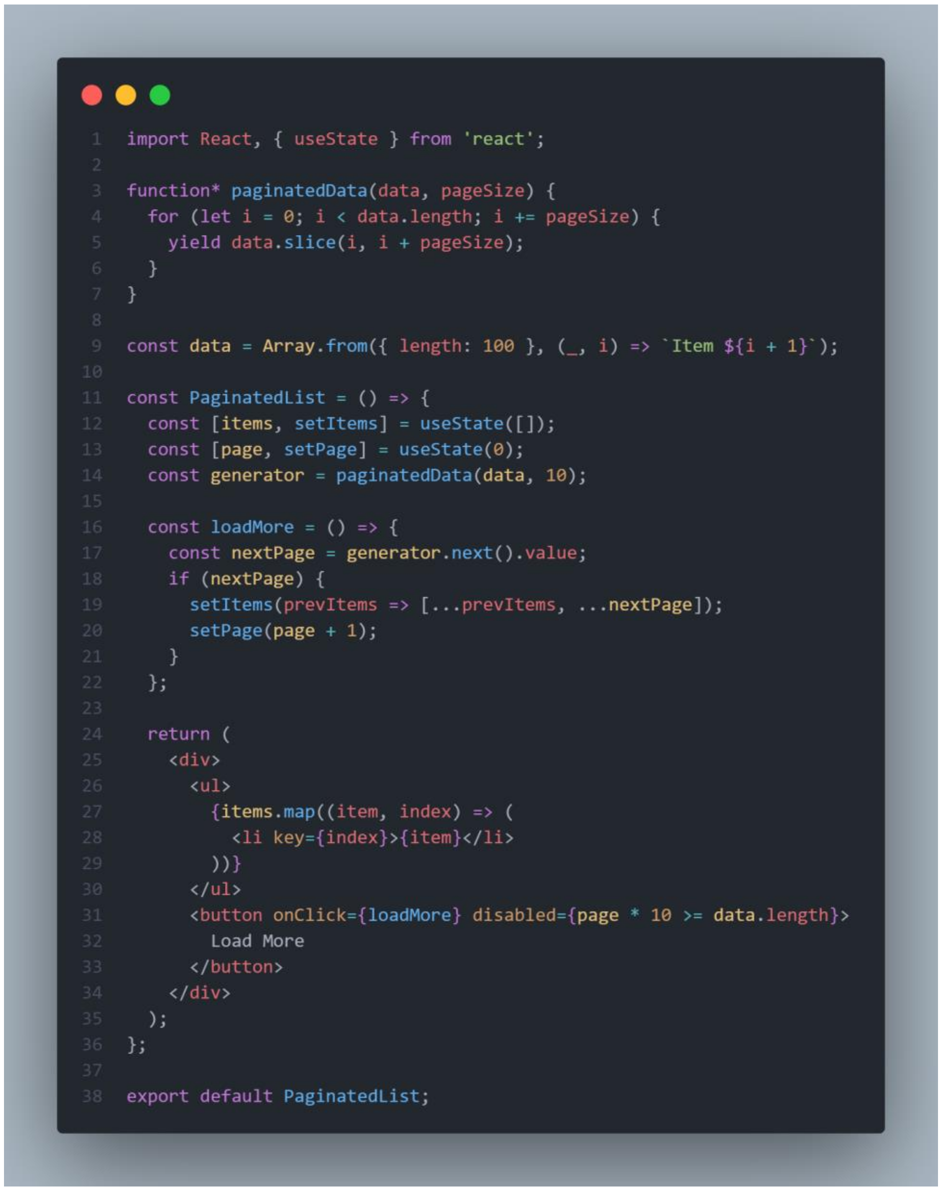
Generators in JS

A generator in JavaScript is a function that can pause its execution and resume later.

Generators are defined using the function* syntax and return a special type of iterator called a generator object which uses the yield keyword to pause and resume execution and they return values incrementally using next() instead of returning all at once as it conforms to both the iterable protocol and the iterator protocol.

"A generator function can be exited and later re-entered, with its context (variable bindings) saved across re-entrances."

Example, pagination: Generators are useful for lazy-loading data. This is example in react, a pagination component where the generator yields a set of items each time a button is clicked.

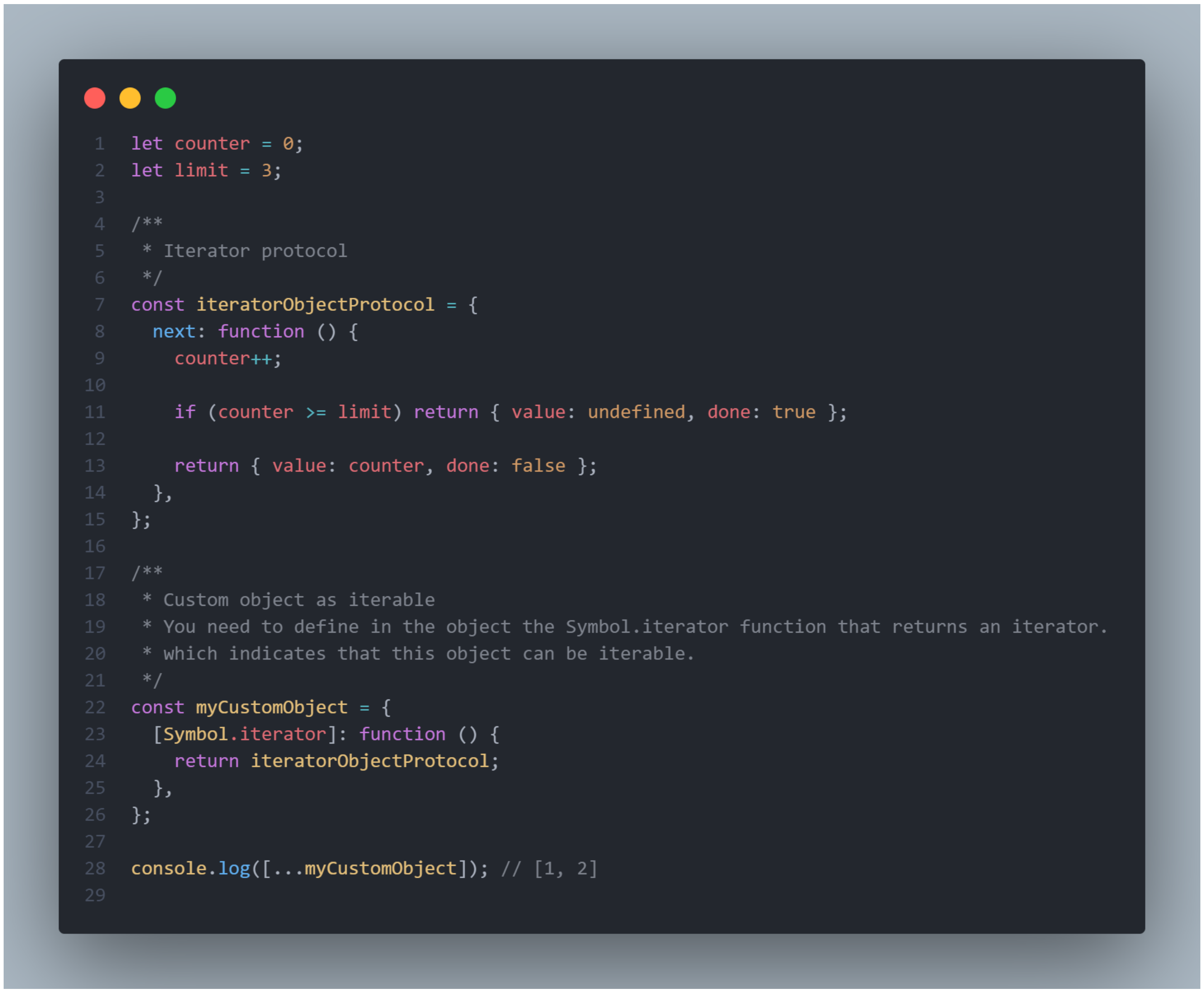


```
import React, { useState } from 'react';

function* paginatedData(data, pageSize) {
  for (let i = 0; i < data.length; i += pageSize) {
    yield data.slice(i, i + pageSize);
  }
}

const data = Array.from({ length: 100 }, (_, i) => `Item ${i + 1}`);

const PaginatedList = () => {
  const [items, setItems] = useState([]);
  const [page, setPage] = useState(0);
  const generator = paginatedData(data, 10);

  const loadMore = () => {
    const nextPage = generator.next().value;
    if (nextPage) {
      setItems(prevItems => [...prevItems, ...nextPage]);
      setPage(page + 1);
    }
  };

  return (
    <div>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={loadMore} disabled={page * 10 >= data.length}>
        Load More
      </button>
    </div>
  );
};

export default PaginatedList;
```

Iterator and Iterable in JS

The iterator protocol defines a standard to produce a sequence of values. This protocol is really an object that contains next() method and this one returns another object with two properties, value and done. Similar to this which is an iterator that returns a counter, the important is that an iterator is an object which returns another object with two properties, value and done.

The iterable protocol indicates that an object is iterable and uses the iterator protocol in it. so, you can iterate it using, for example, for…of and the spread operator ([…obj]). You need to implement the constant Symbol.iterator, this method needs to return an iterator protocol.



```javascript
let counter = 0;
let limit = 3;

/**
 * Iterator protocol
 */
const iteratorObjectProtocol = {
  next: function () {
    counter++;

    if (counter >= limit) return { value: undefined, done: true };

    return { value: counter, done: false };
  },
};

/**
 * Custom object as iterable
 * You need to define in the object the Symbol.iterator function that returns an iterator.
 * which indicates that this object can be iterable.
 */
const myCustomObject = {
  [Symbol.iterator]: function () {
    return iteratorObjectProtocol;
  },
};

console.log([...myCustomObject]); // [1, 2]

```